VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO PARALELNÍ ZPRACOVÁNÍ URL DOTAZŮ

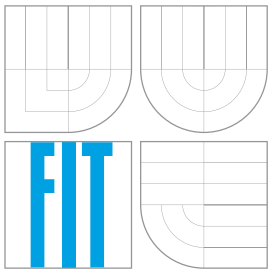
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN HORÁČEK

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO PARALELNÍ ZPRACOVÁNÍ URL DOTAZŮ

SYSTEM FOR PARALLEL PROCESSING OF URL REQUESTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN HORÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAROSLAV RÁB
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií
Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Jan Horáček**
Obor: Informační technologie
Téma: **Systém pro paralelní zpracování URL dotazů**
Kategorie: Počítačové sítě

Pokyny:

1. Seznamte se s knihovnou libCURL, rozhraním pro odesílání URL
2. Navrhněte model systému pro paralelní zpracování URL dotazů
3. Zabezpečení proti výpadku systému, logování do syslogu
4. Navržený model implementujte v jazyce C podle ANSI normy s podporou instalace pomocí autotools
5. Proveďte test systému v reálném provozu
6. Zhodnoťte dosažené výsledky

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Práce pojednává o naprogramování vícevláknové aplikace, jejímž cílem je zpracovávat požadavky pro volání `URL`. Aplikace byla naprogramována v `ANSI` normě jazyka `C`. Běží jakožto samostatný `démon`, který musí v maximální možné míře umožňovat dohledání požadavků, které se nepovedlo zpracovat. K volání `URL` využívá knihovnu `libCURL` a nainstalování na stanici balíku `autotools`.

Klíčová slova

URL, libCURL, UNIX, ANSI, démonizace procesu, lockfile, chroot, HTTP, HTTPS, autotools, syslog, PID, vlákno, read-write zámky

Abstract

The work treats of programming multithreaded application, focused on processing the demands for `URL` calling. The application was programmed in the `ANSI` norm of the language `C`. It runs as independent `daemon`, which must faciliate, in maximum possible scale, searching for demands, which weren't successfully processed. For `URL` calling it uses the library `libCURL` and installing in station of package `autotools`.

Keywords

URL, libCURL, UNIX, ANSI, process daemonizing, lockfile, chroot, HTTP, HTTPS, autotools, syslog, PID, thread, read-write locks

Citace

Jan Horáček: Systém pro paralelní zpracování URL dotazů, bakalářská práce, Brno, FIT VUT v Brně, 2007

Systém pro paralelní zpracování URL dotazů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jaroslava Rába. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Horáček
14. května 2007

Poděkování

Především bych chtěl poděkovat firmě *Mobilbonus s.r.o.*, která preferovala `open-source` model vývoje a tím mohla být tato práce zveřejněna. Dále všem spolupracovníkům této firmy za pomoc při řešení některých problémů.

© Jan Horáček, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**
1.1 Použité zvýraznění v textu - legenda 4

2 Popis práce **5**
2.1 Krok po kroku 5
2.2 Možnosti volání 5

3 Vstupní část serveru **7**
3.1 Implementace 7

4 Démonizace procesu **9**
4.1 Motivace 9
4.2 Vlastní implementace 9
4.3 Kontrola jedné instance programu 10

5 Bezpečnost a zabezpečení proti výpadkům **11**
5.1 Motivace 11
5.2 Změna root adresáře 11
5.3 Změna uživatele 11
5.3.1 Implementace 12
5.3.2 Poznámky 12
5.4 Logování pomocí systému syslog 12
5.5 Způsob uložení jednotlivých požadavků 12
5.5.1 Proč soubory? 13
5.5.2 Pojmenování souboru v adresáři 13

6 Vícevláknové programování **14**
6.1 Přístup ke sdílenému zdroji 14
6.1.1 Funkce, typy a makra uvozené “`pthread_mutex_...`” 14
6.1.2 Read-Write zámky 15
6.2 Omezení maximálního počtu vláken 15

7 Konfigurace serveru **17**
7.1 Lexikální analyzátor 17
7.2 Syntaktický analyzátor 17
7.3 Struktura uchovávané konfigurace v paměti 17

8 libCURL **19**
8.1 O knihovně libCURL . . . 19
8.2 Způsob používání knihovny . . . 19
8.3 Nastavení parametrů pro volání dané URL . . . 20
8.3.1 Parametr CURLOPT_URL . . . 20
8.3.2 Parametr CURLOPT_FILE . . . 20
8.3.3 Parametr CURLOPT_TIMEOUT . . . 20
8.3.4 Parametr CURLOPT_NOSIGNAL . . . 20
8.3.5 Parametr CURLOPT_PORT . . . 20
8.3.6 Parametr CURLOPT_USERAGENT . . . 20

9 Balík programů autotools **21**
9.1 Program autoscan . . . 21
9.2 Program autoheader . . . 21
9.3 Vybraná makra programu autoheader . . . 21
9.3.1 Makro AC_PREREQ . . . 22
9.3.2 Makro AC_INIT . . . 22
9.3.3 Makro AC_COPYRIGHT . . . 22
9.3.4 Makro AC_CONFIG_SRCDIR . . . 22
9.3.5 Makro AC_CONFIG_HEADER . . . 22
9.3.6 Makro AC_PREFIX_DEFAULT . . . 22
9.3.7 Makro AC_CHECK_LIB . . . 22
9.3.8 Makra AC_CHECK_HEADER a AC_CHECK_HEADERS . . . 22
9.3.9 Makro AC_CHECK_FUNCS . . . 22
9.3.10 Makro AC_SUBST . . . 22
9.3.11 Makro AC_CONFIG_FILES . . . 22
9.3.12 Makro AC_OUTPUT . . . 23
9.3.13 Makra AC_MSG_ERROR a AC_MSG_NOTICE . . . 23
9.4 Program autoconf . . . 23

10 Kompilace, spuštění a testování **24**
10.1 Kompilace a instalace . . . 24
10.2 Testování . . . 25

11 Závěr **26**

12 Příloha A **28**

13 Příloha B **30**

Kapitola 1

Úvod

Tento software byl vyvinut pro firmu *Mobilbonus s.r.o.* v Jihlavě. Bylo potřeba oddělit jednovláknovou část systému, která generuje požadavky na komunikaci v síti internet, od části, která se pokoušela `URL` volat. Byla zvolena implementace v podobě malého serveru, který naslouchá na určitém portu a sám průběžně požadované `URL` volá. Na jeho vstupu přijme pouze co za `URL` má zavolat a ihned vrací informaci, že požadavek přijal. `URL` později sám zavolá. Tímto způsobem nemusí jednovláknová část systému čekat na příchozí data z `URL` a pozastavovat na tuto dobu svoji činnost.

Jedním z požadavkům bylo, aby aplikace zůstala v čisté `ANSI` normě jazyka `C`. To přineslo mnohé problémy, z nichž se většinu podařilo vyřešit.

V jednotlivých kapitolách tohoto textu bude pojednáno o tom, jak vytvoříme samostatně běžící server, jak zabezpečit systém proti výpadkům a jak řídit běh více vláken pracujících se stejným zdrojem. Dále pak, jak server načítá svou konfiguraci. Nemalá část pojednává o knihovně `libCURL`, která byla vybrána z několika důvodů, oproti ostatním knihovnám s podobým zaměřením. Je zde obsažena i kapitola řešící problém automatické instalace s využitím balíku nástrojů `autotools`. V samotném závěru jsou obsaženy kapitoly zabývající se testováním a shrnutím dosažených výsledků.

Tyto kapitoly byly voleny s ohledem na subjektivní zajímavost v projektu. Pro další označení v textu dostal server jméno *url_caller*.

1.1 Použité zvýraznění v textu - legenda

- `<soubor.h>`
  Systémový hlavičkový soubor.
- "`soubor.h`"
  Vlastní zdrojový nebo hlavičkový soubor. Soubory jsou umístěny v adresáři projektu.
- *main()*, *main()*[1]
  Funkce - s případným komentářem.
- (`*char`)
  Datový typ.
- `slovo`
  Strojopisným písmem označíme makra, definice v kódu a významná slova.
- *komentář*[2]
  Komentář vztahující se ke konkrétnímu slovu.
- http://www.google.com/
  Internetový odkaz.

[1] funkce s komentářem
[2] komentované slovo

Kapitola 2

Popis práce

Zde bude uvedeno, jak daný program funguje, jaké jsou jeho součásti a jejich vazby.

2.1 Krok po kroku

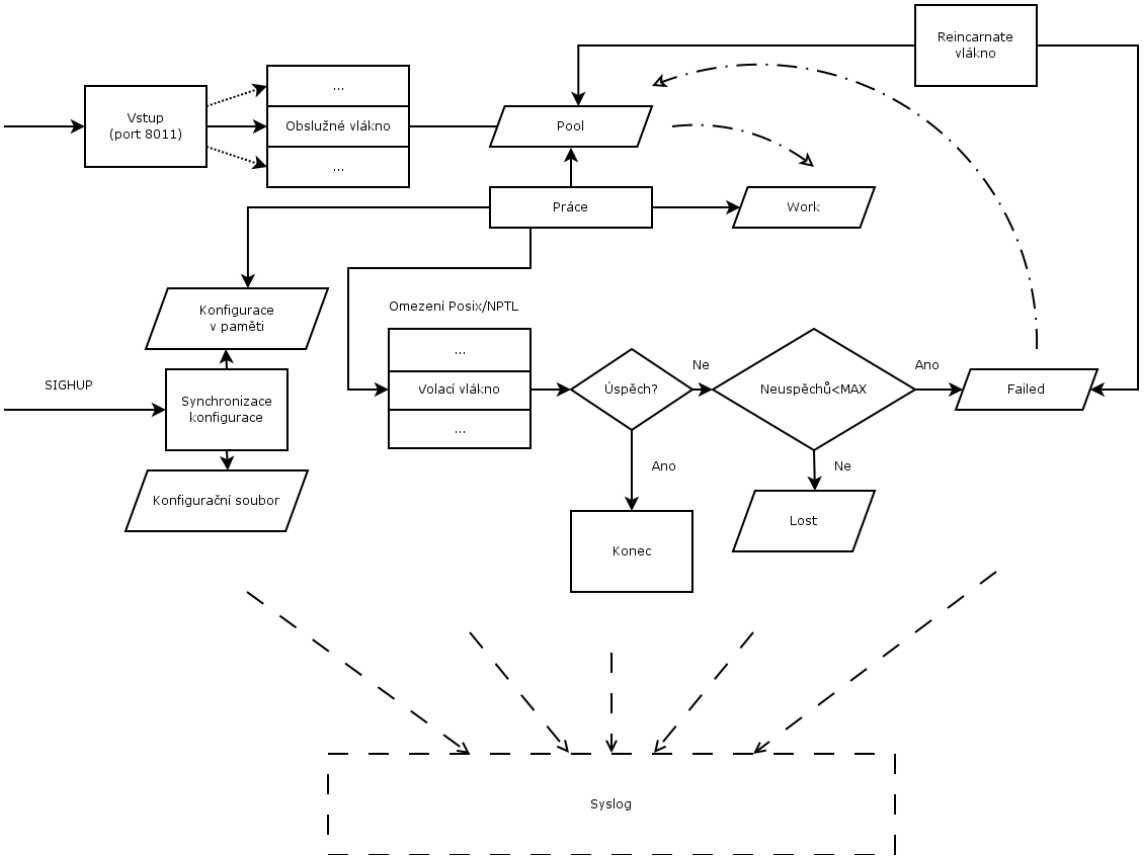
Server naslouchá na zvoleném portu a přebírá zadané požadavky. Pokud daný požadavek přijme a vše je v pořádku, vrací nazpět řetězec "`1\n`"[1]. Volanou `URL` zapíše na samostatný řádek do souboru v adresáři `pool`. Dále připojí další řádek identifikující počet pokusů o volání dané `URL` - na počátku 0. Další vlákno v sekundových intervalech kontroluje adresář `pool` a vybírá z něj jednotlivé soubory ke zpracování. Zpracovávaný soubor přesune do adresáře `work` a zde z něj vyčte, jakou `URL` má volat a dále kolikrát již byla volána. Pokusí se zavolat danou `URL` a připojí její obsah na konec souboru. V případě úspěchu, pokud je v konfiguraci uvedeno, porovná obsah s porovnávacím řetězcem a označí volání jako úspěšné/neúspěšné. V případě neúspěchu volání, zvedne počítadlo pokusů o volání a přesouvá soubor do adresáře `failed`. Zde je v pravidelných intervalech kontrolován obsah adresáře a zpětně jsou soubory přesouvány do adresáře `pool`. Pokud byl nastaven maximální počet pokusů o znovu volání `URL` a je tento počet překročen, je soubor namísto do adresáře `failed`, přesunut do adresáře `lost`. Při každém přesunu je nutno v cílovém adresáři najít unikátní jméno cílového souboru. Vše je možno vidět na obrázku 2.1.

2.2 Možnosti volání

Server *url_caller* umožňuje spojení `URL` běžným `HTTP` protokolem, nebo přes šifrované `HTTPS`. Zde se vyskytl problém. V době vývoje tohoto systému nebyly k dispozici platné certifikáty podepsané naší certifikační autoritou. Proto je prozatím implementováno pouze ověřování platnosti certifikátu.

V konfiguraci může být uvedeno, aby `URL` nevolal na standardní port 80[2], ale na některý jiný.

[1] odřádkovaná jednička
[2] v případě `HTTPS` port 443



Obrázek 2.1: **Základní části systému**
Na obrázku je vidět základní rozdělení a spolupráce jednotlivých bloků.

Kapitola 3

Vstupní část serveru

Aby mohly externí programy našemu serveru zasílat požadavky, musíme učinit několik kroků. Tyto kroky budou rozebrány dále v této kapitole. Nejvíce bylo čerpáno z knihy *UNIX Network Programming* [8].

3.1 Implementace

Komunikace byla vystavěna na vrstvě `TCP/IP`. To mimo jiné znamená, že vrstva `TCP` (Transmission Control Protocol) se postará o navázání komunikace[1], její ukončení a dále také o správné řízení komunikace. Ta spočívá ve správném řazení `packetů`, kontrole zda dané `packety` dorazily v pořádku, ochraně proti zahlcení přijímací strany atd. Zavoláním funkce *run_server()* implementevané v souboru "`server.c`" spustíme server na určitém portu. Jako základní nastavení byl zvolen port 8011.

Nejprve vytvoříme *socket*[2] stejnojmennou funkcí *socket()*[3] nacházející se v systémové knihovně `<sys/socket.h>`. Jako první parametr zvolíme `PF_INET`, který značí, že chceme komunikovat protokolem `IPv4`. V minulosti bylo rozlišováno mezi hodnotou `AF_INET` a `PF_INET`. Jednalo se o rozdíl v podpoře více adres, ovšem v dnešní době již rozdíl v hodnotách není. Druhý parametr zadáme `SOCK_STREAM` říkající, že se jedná o tok dat. Poslední parametr `IPPROTO_TCP` označuje komunikaci na vrstvě `TCP`.

Do datové struktury "`struct sockaddr_in`" vložíme za `sin_family` hodnotu `PF_INET`, za `sin_addr.s_addr` hodnotu `INADDR_ANY` značící, že si přejeme komunikovat přes všechna rozhraní. Do `sin_port` vložíme, na jakém portu chceme aby server běžel. Port zadáme jako běžné číslo funkci *htons()* ze souboru `<arpa/inet.h>`. Ta pouze zamění pořadí bytů z *LSB*[4] na *MSB*[5].

Funkcí *bind()*[6] svážeme lokální adresu se `socketem`. Prvním parametrem je námi vytvořený `socket`, druhým ukazatel na nastavenou strukturu "`struct sockaddr_in`", kterou je nutno přetypovat na "`struct sockaddr*`" a třetím je velikost této struktury. Nakonec funkcí *listen()*[7] začneme naslouchat na námi zvoleném portu. Druhým parametrem funkce *listen()* je hodnota, označující délku fronty, kterou kernel vytváří pro příchozí spojení, která

[1] tzv. three-way handshake
[2] nejlépe přeložitelný výraz do češtiny by mohl být „zástrčka“
[3] int socket(int domain, int type, int protocol);
[4] Least Significant Byte
[5] Most Significant Byte
[6] int bind(int sockfd, const struct sockaddr *my_addraddrlen);
[7] int listen(int sockfd, int backlog);

backlog	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

Tabulka 3.1: **Backlog**
V tabulce závislost počtu spojení ve frontě na hodnotě backlog

ještě nebyla obsloužena. Z této hodnoty můžeme převodní tabulkou pro konkrétní operační systém určit maximální počet spojení ve frontě. V tabulce 3.1 je dobré si všimnout, že například u `BSD/OS` je počet spojení roven backlog + 1. Maximální počet těchto spojení je omezen definicí `SOMAXCONN` v `<sys/socket.h>`, která je obvykle nastavena na 128. Hodnota backlog byla zvolena 100.

Funkce *accept()*[8] v blokujícím režimu čeká na příchozí spojení. Pokud se spojení naváže, pak vrací nový `deskriptor` vytvořený kernelem. Zde již zbývá vytvořit nové vlákno, které spojení obslouží. Toto nové vlákno začíná svou práci ve vlastní funkci *handle_connect()*[9].

[8] int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
[9] void *handle_connect(void *arg)

Kapitola 4

Démonizace procesu

Pokud chceme program nechat běžet na pozadí tak, aby byl odpojen standardní vstup a výstup terminálu, musíme proces démonizovat. Tento proces taktéž musí být, ve stromové stuktuře procesů, závislý pouze na *init*[1] procesu.

4.1 Motivace

V hlavičkovém souboru `<unistd.h>` sice existuje funkce *daemon()*[2], ale ta je bohužel zamaskovaná pomocí `__USE_BSD` nebo `__USE_XOPEN`. Chceme-li zůstat v čisté `ANSI` normě, musíme si tuto funkci ručně naprogramovat.

4.2 Vlastní implementace

Vlastní implementace démonizace procesu se nalézá v hlavičkovém souboru "`daemonize.h`" jakožto funkce *daemonize()*[3]. Je složena z několika postupných kroků, které je nutno vykonat. Jedná se především o:

- *Fork*[4] procesu a ukončení původního
- Vytvoření nového sezení - přímým rodičem se stává init
- Ignorace signálů `SIGTSTP`, `SIGTTOU`, `SIGTTIN`, které přicházejí z terminálu ve kterém byl program spuštěn a uzavření vstupů `stdin`, `stdout`, `stderr`
- Reakce na signál `SIGTERM`, pro ukončení programu

Mimo tyto celkem běžné kroky je ve funkci *daemonize()* navíc obsažena kontrola běhu pouze jednoho serveru. Je kontrolována přes soubor, ve kterém je obsaženo *PID*[5] běžícího souboru. Pokud je tento speciální soubor přítomen, nově spouštějící se program skončí. Tomuto souboru se běžně říká `lockfile`. Dále je zde zajištěna obsluha signálu `SIGHUP`, při jehož vyvolání program znovu načte konfiguraci.

[1] proces, který nemá rodiče
[2] int daemon(int nochdir, int noclose);
[3] void daemonize();
[4] Rozdvojení procesu na dva samostatně běžící
[5] Process ID - unikátní číslo označující proces

4.3 Kontrola jedné instance programu

Jak již bylo zmíněno, je pro tuto funkci využit soubor `lockfile`. Soubor otevřeme pro zápis. Uzamkneme ho s využitím systémové funkce *flock()*[6] z hlavičkového souboru `<sys/file.h>`. Jako parametry zvolíme daný soubor a dále kombinaci `LOCK_EX | LOCK_NB`. `LOCK_EX`[7] je zde kvůli vyloučení zamknutí souboru jiným procesem. Dále je dobré, dle dokumentace, zadat operaci jako neblokující pomocí `LOCK_NB`[8]. To zajistí v případě již zamknutého souboru, vrácení se zpět do programu s tím, že se daná operace nezdařila. V tom případě program ihned skončí a nečeká na uvolnění souboru. Pokud se povedlo si soubor pro sebe uzamknout, vepíšeme do něj své PID.

[6] int flock(int fd, int operation);
[7] z anglického exclusive
[8] non blocking

Kapitola 5

Bezpečnost a zabezpečení proti výpadkům

U každého programu je vždy třeba se zabývat otázkou bezpečnosti. Zde budou rozebrány dvě běžné metody s cílem zabezpečení daného programu. Dále metody, pro snadné dohledání stavu systému před poruchou s cílem minimální ztráty dat.

5.1 Motivace

Každý systém má své bezpečnostní chyby. Pro jejich minimalizaci se používá mnoho metod. V serveru *url_caller* bylo původně počítáno s využitím dvou metod. Bohužel metoda využívající funkci *chroot()*[1] být použita nemohla. Byla alespoň implementována změna uživatele, pod kterým server běží. Dále bylo implementováno *logování*[2] pomocí systému `syslog` a navržen systém úchovy zpracovávaných dat.

5.2 Změna root adresáře

Root adresář je ve stromové struktuře vždy nejvýše. Po přidělení root adresáře již nemůže daný proces nikdy vystoupit v adresářové struktuře výše. Bohužel je opět v hlavičkovém souboru `<unistd.h>` zamaskována funkce *chroot()* pomocí testu definice `__USE_BSD` nebo `__USE_XOPEN`. Zde je nutno říci, že tato metoda je z pohledu vlastní implementace těžko realizovatelná. Proto bylo upuštěno od myšlenky změny root adresáře přímo v programu. Lze ovšem využít stejnojmeného programu, což je taktéž i doporučením.

5.3 Změna uživatele

Daný proces vždy běží pod určitým uživatelem a skupinou. Ti mají svá oprávnění. Nejvyšší oprávnění má uživatel root[3] a ten se také může přepnout na jiného uživatele s nižšími právy.

[1] int chroot(const char *path);
[2] záznam určité informace
[3] na některých systémech jiný

5.3.1 Implementace

Nalézá se přímo v souboru "`main.c`" pod funkcí *change_priv()*. Zde nejprve zjistíme z konfiguračního souboru, pod kterým uživatelem a skupinou má proces běžet. Poté, s pomocí systémových funkcí *getpwnam()* a *getgrnam()*, dostaneme ukazatel na strukturu uchovávající informace o nich. Z této struktury potřebujeme znát pouze ID uživatele a skupiny. Nastavíme skupinu, pomocí funkce *setgid()*, a poté uživatele funkcí *setuid()* (v tomto pořadí). Následně zkusíme nastavit zpět ID uživatele na 0 (root), což by systém neměl dovolit. Protože v některých starých jádrech řady 2.2 byl tento závažný bug, tak raději otestujeme i tuto možnost.

5.3.2 Poznámky

U funkcí *getpwnam()* a *getgrnam()* byl sledován zvláštní jev. Při kontrole uvolňování paměti programem *valgrind*[4] bylo zjištěno, že tyto dvě funkce vrací ukazatel na paměť, kterou již nelze uvolnit. Funkce *getpwnam_r()* z `<pwd.h>` a *getgrnam_r()* z hlavičkového souboru `<grp.h>` vytvářejí struktury s daty o uživatelých a skupinách v paměti přidělené programu[5]. Jsou bouhužel zamaskovány testem definice `__USE_BSD` nebo `__USE_XOPEN`. Po několika hodinách se tento problém jevil jako nevyřešitelný (podařilo se uvolnit pouhou část této paměti). Je dobré zde zmínit, že velmi podobně jsou napsány funkce na změnu uživatelských práv v mnoha webových serverech. Pro ukázku server `Apache` [3] či `lighttpd` [4] se naprosto o žádné uvolňování této paměti nesnaží.

5.4 Logování pomocí systému syslog

Cílem je zaznamenat určitou informaci o aktuálním stavu systému. Může se jednat o registraci nějakého chybového stavu, či prostou informaci např. o příchozím spojení. Tento systém záznamu informace (též logování) je možné distribuovat do různých míst podle aktuálního nastavení. Jednotlivé zprávy mají svůj typ a původ informace, doprovázený jejím obsahem. Můžeme například nastavit, aby se běžné zprávy naprosto ignorovaly a chybové zprávy posílaly na jiný server. Záleží pouze na nastavení logovacího démona (`syslog`, `syslog-ng`, ...).

Vše zajišťuje funkce *syslog()*[6] z hlavičkového souboru `<sys/syslog.h>`. Jako původ informace zvolíme démon - pomocí `LOG_DAEMON`. Typ informace zvolíme buď `LOG_INFO`, a nebo `LOG_ERR` podle toho, zda se jedná o běžnou informaci či chybový stav. Vzorové zavolání pak může vypadat například:

```
syslog(LOG_ERR|LOG_DAEMON, ‘‘Aplikace spadla’’);
```

5.5 Způsob uložení jednotlivých požadavků

Pro uložení požadavků využívám obyčejných souborů. Jejich obsah se skládá z prvního řádku obsahující `URL` a na dalším řádku uvedeným počtem pokusů o volání. Od další řádky se nachází stažený obsah, který v případě potřeby zkontrolujeme, zda odpovídá našim požadavkům. Jsou postupně přesouvány mezi adresáři `pool`, `work`, `failed` a `lost`.

[4] trasovací program pro sledování přidělené a uvolněné paměti
[5] čímž by se dala paměť uvolnit
[6] void syslog(int priority, const char *format, ...);

5.5.1 Proč soubory?

Je to způsob perzistentního uložení dat pro případ pádu aplikace. Jelikož jsou tyto soubory obsahově velmi malé a jejich počet může být velmi vysoký, je pro jejich uložení doporučeno použít vhodný souborový systém. Jako vhodným se zdá být `ReiserFS`.

Jednoduchou operací přesunutí souboru mezi jednotlivými adresáři dáváme jaksi najevo změnu stavu daného požadavku. Adresář `pool` slouží jako kontejner požadavků, které čekají na obsloužení. Adresář `work` obsahuje právě zpracovávané požadavky. Adresář `failed` požadavky, které se nepovedlo zpracovat s tím, že se o jejich obsloužení v budoucnu ještě pokusí. A konečně adresář `lost` schraňuje všechny požadavky, které se nepovedlo vyřídit a u nichž se již o opětovné obsloužení nepokoušíme.

Operace pro přesunutí spočívá v pouhém přesunutí *i-nodu*[7] z jednoho adresáře do druhého a díky tomu je velmi rychlá. Zavoláním funkce *rename()*[8] ze systémové knihovny `<stdio.h>` provedeme přesun souboru.

5.5.2 Pojmenování souboru v adresáři

Je potřeba, aby každý soubor v adresáři měl své jednoznačné jméno. Pokud chceme vytvořit nový soubor, musíme toto jméno nějakým způsobem „vymyslet“. Vzhledem k tomu, že funkce *tmpnam()* ze systémové knihovny `<stdio.h>` má omezený počet volání definovanou hodnotou `TMP_MAX`, musíme naprogramovat vlastní funkci. Hodnota sice bývá na většině UNIXových systémů velmi vysoká, ale stále není nekonečná.

Funkce je pojmenována jako *make_tmp_name()*[9] v souboru “`dirs.c`” a přebírá dva parametry. První je adresář, ve kterém se bude snažit nové jméno souboru hledat a druhým je, kolik znaků má jméno obsahovat. Nejprve náhodně vygeneruje jméno o požadované délce znaků a funkcí *access()*[10] z `<unistd.h>` zkontroluje, zda soubor existuje. Takto hledá v cyklu do doby, dokud nenajde jméno souboru, který neexistuje. Aby se program nezacyklil, je počet cyklů omezen pomocí definované hodnoty `MAX_TRYING`. Bezprostředně po této funkci vždy následuje vytvoření souboru a tento blok musí být uzavřený v kritické sekci. Jinak by se mohlo stát, že dvě vlákna vygenerují stejné jméno souboru, ale díky prozativní neexistenci souboru vrátí *make_tmp_name()* do obou vláken stejné jméno souboru.

[7] datová struktura obsahující metadata o souborech a adresářích
[8] int rename(const char *oldpath, const char *newpath);
[9] char * make_tmp_name(const char * directory, int length);
[10] int access(const char *pathname, int mode);

Kapitola 6

Vícevláknové programování

Tato kapitola se zabývá programováním aplikací složených z více vláken. Řeší se především přístup vláken ke sdílenému zdroji a dále komunikace mezi vlákny. Dobrou literaturaturou na toto téma[1] se zdá být kniha *Advanced UNIX Programming* [6] a část je obsažena i v knize *UNIX Network Programming* [7]. Případně se dá mnohé zjistit nahlédnutím do zdrojových kódů `kernelu` [5].

6.1 Přístup ke sdílenému zdroji

Pokud se k danému zdroji snaží přistoupit více vláken současně a tyto data se potřebují měnit, je nutné přístup a manipulaci s nimi nějakým způsobem řídit. Této části ve zdrojovém kódu říkáme kritická sekce. K řízení přístupu do kritické sekce se využívá kromě jiných:

- `mutex`
- `semaphore` - semafor

Semafory, vzhledem k možnosti `deadlocku` a podivné absenci při použití `ANSI` na linuxové distribuci `Debian`, nebyly použity.

6.1.1 Funkce, typy a makra uvozené "`pthread_mutex_...`"

Nejprve budeme potřebovat statickou proměnnou typu `pthread_mutex_t`. Tu můžeme inicializovat buď zavoláním funkce *pthread_mutex_init()*[2], nebo ihned s pomocí makra `PTHREAD_MUTEX_INITIALIZER`. Tuto proměnnou pak při vstupu do kritické sekce předáme funkci *pthread_mutex_lock()*[3] a při výstupu z kritické sekce funkci *pthread_mutex_unlock()*[4]. Tyto funkce pocházejí ze systémové knihovny `<pthread.h>`. Uvedu jednoduchý příklad:

Příklad ošetření kritické sekce

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;


void foo()
{
```

[1] i jiná témata
[2] int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
[3] int pthread_mutex_lock(pthread_mutex_t *mutex);
[4] int pthread_mutex_unlock(pthread_mutex_t *mutex);

```
  /* Nějaký kód */
  pthread_mutex_lock(&foo_mutex);
    /* Kritická sekce - přístup ke sdílenému zdroji */
  pthread_mutex_unlock(&foo_mutex);
  /* Nějaký kód */
}
```

6.1.2 Read-Write zámky

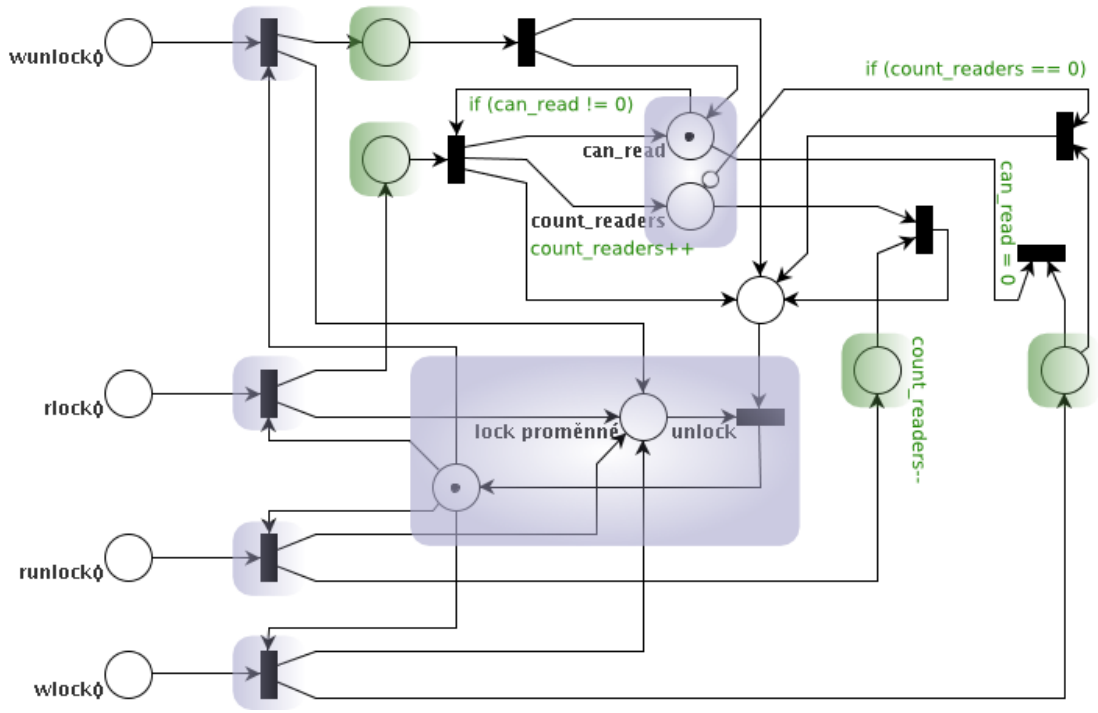
Jedná se o přístup ke sdílenému zdroji, ke kterému většinu času přistupují vlákna s cílem číst data (readers - čtenáři). Ovšem pokud tyto data chceme změnit (writers - písaři), potřebujeme zdroj uzamknout, nová data do něj zapsat a následně znovu zpřístupnit čtenářům. Pro tento účel je v hlavičkovém souboru `<pthread.h>` několik funkcí uvozených "`pthread_rwlock_...`". Bohužel jsou všechny tyto funkce zamaskovány pomocí testu na definici `__USE_UNIX98` nebo `__USE_XOPEN2K`.

Nezbývá, než napsat vlastní implementaci. V souboru "`read_write.c`" je uvedeno několik funkcí, které zabezpečují tuto funkcionalitu. Jedná se o zjednodušené řešení, kdy existuje pouze jeden písař. Funkce *rwlock_rlock()* se pokusí ke zdroji přistoupit jako čtenář. Pokud do zdroje nikdo nezapisuje, nečeká a ihned může číst data. Dále zvedne počet čtenářů v proměnné `count_readers`. Funkce *rwlock_runlock()* označuje konec čtení dat ze sdíleného zdroje a sníží hodnotu `count_readers`. Funkce *rwlock_wlock()* nejprve zajistí, aby všichni čtenáři nemohli přistoupit ke zdroji. K tomu slouží proměnná `can_read` nastavená na 0. Dále počká na všechny čtenáře, kteří začali pracovat se zdrojem před zavoláním *rwlock_wlock()* a práci dokončují. Pokud všichni čtenáři dokončili svou práci, je proměnná `count_readers` nastavena na 0. V tuto chvíli je zdroj uzamknut pro zápis. Funkce *rwlock_wunlock()* uvolní zdroj nastavením `can_read` na 1. Proměnné `can_read` a `count_readers` musí být uzavřeny v kritické sekci. Makro *mwait()* tyto proměnné uzamkne, a tak můžeme zjistit/zapsat jejich hodnoty a makro *msignal()* uvolní pro ostatní vlákna. Na obrázku 6.1 jsou jednotlivé funkce v *Petriho síti*[5].

6.2 Omezení maximálního počtu vláken

Většina dnešních PC má jedno zásadní omezení. Tím je maximální počet současně běžících procesů[6]. Proto je nutné šetřit s těmito prostředky. Situace se mění při použití `NPTL` vláken. V linuxové distribuci `Gentoo`, pro který byl tento server vyvíjen, jsou standardně zvolena `NPTL` vlákna. Pokud se v jiných distribucích (např. `Debian`) nezměnil tento model zprávy procesů na `NPTL`, server se jevil jako značně nestabilní. Stejná situace nastala při nasazení na operační systém `FreeBSD`. Nutno přiznat, že je to asi jedna z hlavních částí práce do budoucna. Na cílové stanici, pro který byl server vyvíjen, se zdá, že by měl být systém stabilní. Více v kapitole **Testování** na straně 25.

[5] Petriho síť je grafická reprezentace diskrétních distribuovaných systémů
[6] běžná hodnota je 4096



Obrázek 6.1: **Read-Write zámky**
Na obrázku zobrazená Petriho síť

Kapitola 7

Konfigurace serveru

Konfigurace serveru je načítána ze souboru "`/etc/url_caller.conf`". Jeho vzorové nastavení je v příloze **B** na straně 30. Při startu serveru se podívá do tohoto souboru a načte ji do paměti synchronizovanou `read-write` zámkem. Nastaví, na jakém portu má server běžet, pod jakým uživatelem a jakou skupinou. Tyto vlastnosti již nemůže z logických důvodů měnit[1]. Při přijetí signálu `SIGHUP` zamkne `read-write` zámkem paměť pro zápis a nahraje do paměti novou konfiguraci. Paměť zpřístupní pro čtení všem ostatním vláknům. Načítání konfigurace je zprostředkováno přes lexikální a syntaktický analyzátor.

7.1 Lexikální analyzátor

Nachází se v souboru "`lex.c`" a jeho hlavičkovém souboru "`lex.h`". Je to klasický stavový automat, jehož účelem je rozdělení textu na části. Určí, zda daná část je číslo, řetězec, klíčové slovo atp. Klíčová slova následně porovnává v souboru "`scann.c`". Jejich definice jsou uvedeny v "`scann.h`". Je volán syntaktickým analyzátorem, aby mu poskytl další část vstupu s obsahem a označením typu.

7.2 Syntaktický analyzátor

Syntaktický analyzátor se nachází spolu se sémantickými akcemi v souboru "`options.c`". Jedná se opět o stavový automat. Jeho úkolem je kontrola, zda na sebe lexikální jednotky navazují v pořadích, která jsou možná. Každé této možné kombinaci přiřadí sémantickou akci. Jako příklad uvedu:

„Za klíčovým slovem `TIMEOUT` následuje číslo. Pak sémantickou akcí bude přiřazení atributu maximální doby čekání na daný počet sekund.“

7.3 Struktura uchovávané konfigurace v paměti

Konfigurace je rozdělena do několika částí:

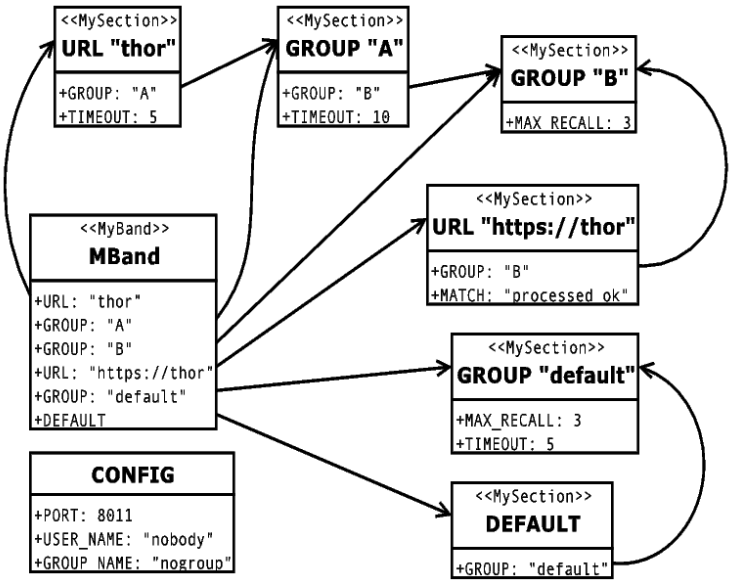
- Sekce (`URL`) - Počáteční část řetězce identifikující `URL`.
- Skupiny (`GROUP`) - Do skupin můžeme přiřadit jednotlivé sekce.
- Default (`DEFAULT`) - Speciální typ sekce. Pokud neexistuje vhodná sekce, použije tuto.

[1] nemá již dostatečná oprávnění

- Config (`CONFIG`) - Obsahuje nastavení serveru, jako je port a uživatel pod kterým má běžet.

Do dané sekce patří každá `URL`, která začíná řetězcem uvedeným v konfiguraci. Chceme-li například zavolat http://thor/1.php i http://thor/2.php se stejnou konfigurací, zavedeme sekci "`http://thor/`". Sekce mohou být zařazeny do více skupin současně. Skupiny mohou být taktéž zařazeny ve skupinách a tvořit tak stromovou strukturu. Zde je nutné zmínit **upozornění**, že zde může dojít k nekonečné rekurzi, protože program nijak nekontroluje cyklické vazby.

Části `URL`, `GROUP` a `DEFAULT` jsou uchovávány v datové struktuře `MySection`, jejíž definice se nachází v hlavičkovém souboru "`options.h`". Všechny tyto části jsou jako celek drženy ve struktuře `MyBand`, na kterou je aplikován zmiňovaný `read-write` zámek. Část `CONFIG` je uchovávána samostatně a její změna na již běžící server nemá vliv. Na obrázku 7.1 je zobrazen příklad. Seznam jednotlivých atributů lze nalézt v programové dokumentaci.



Obrázek 7.1: **Uložení konfigurace v paměti**
Na obrázku zobrazený vzorový příklad

Kapitola 8

libCURL

Tato kapitola se zabývá knihovnou `libCURL`, která poskytuje rozhraní pro *HTTP/HTTPS*[1] komunikaci. Je stáhnutelná na svých domácích stránkách [1]

8.1 O knihovně libCURL

Tato knihovna je šířena pod licencí *MIT/X*[2], jejíž přesné znění lze nalézt na internetu [2]. Jak je na úvodních stránkách tohoto projektu uvedeno, jedná se o multiplatformní knihovnu portovanou téměř pro všechny hlavní operační systémy. Podporuje mnoho komunikačních protokolů (`FTP`, `FTPS`, `HTTP`, `HTTPS`, `SCP`, `SFTP`, `TFTP`, `TELNET`, `DICT`, `FILE` a `LDAP`). Pro využití v bakalářské práci postačil `HTTP` a `HTTPS` protokol. Tato knihovna má také další rozhraní pro jiné programovací jazyky. Knihovnu využívá mnoho známých programů i knihoven - zmiňme například `libTorrent`, `OpenOffice.org`, `Xen` a mnoho dalších. Hlavní výhodou, oproti některým jiným knihovnám je, že je *thread-safe*[3].

8.2 Způsob používání knihovny

Nejprve celou knihovnu inicializujeme zavoláním *curl_global_init()*. Knihovna podporuje 3 základní modely rozhraní:

- `easy` interface
- `multi` interface
- `shared` interface

Jedná se v podstatě o rozlišení práce při volání více URL současně. `Easy` rozhraní pracuje jakožto jedno volání - jeden *handle*[4]. Toto rozhraní bylo použito v bakalářské práci. Za zmínku stojí uvést rozhraní `multi`. Rozhraní pracuje pouze s jedním `handle` i pro více volaných URL. Ovšem toto rozhraní má jednu podstatnou vadu na kráse - nepodporuje `proxy` spojení. Rozhraní `shared` je zjednodušené `multi`, u kterého mají všechny volané URL nastaveny stejné dodatečné vlastnosti, čímž se pro projekt stalo nepoužitelným.

[1] poskytuje rozhraní i pro mnoho dalších protokolů
[2] jedna z nejvolnějších licencí, přibližně srovnatelná s třířádkovou verzí BSD licence
[3] vlákna jedné aplikace se nemusí zabývat kritickými sekcemi
[4] ukazatel na strukturu uchovávající informace o spojení

Jak již bylo zmíněno, použijeme rozhraní `easy`, ve kterém získáme `handle` díky zavolání *curl_easy_init()*. Otevřeme soubor pro zápis (pro ukládání stažených dat). Následuje sekvence nastavení konkrétních vlastností pro volání URL. Po nastavení všech vlastností zavoláme danou URL funkci *curl_easy_perform()*[5], která vrací status kód dané operace. Pokud je vrácený status kód `CURLE_OK`, vše proběhlo v pořádku. Daný handle uvolníme zavoláním *curl_easy_cleanup()*. Při ukončování programu „vyčistíme“ knihovnu zavoláním *curl_global_cleanup()*.

8.3 Nastavení parametrů pro volání dané URL

Nastavování parametrů volání URL je zprostředkováno zavoláním funkce *curl_easy_setopt()*[6]. Je definováno několik hodnot `CURLoption`, které nastavují různé části. V dalších podkapitolách se budu zabývat pouze použitými nastaveními.

8.3.1 Parametr CURLOPT_URL

Jako další parametr se bude brát řetězec (`*char`) identifikující volanou URL.

8.3.2 Parametr CURLOPT_FILE

Nastavením `CURLOPT_FILE` se přiřadí soubor (`*FILE`), do kterého se uloží stažený obsah.

8.3.3 Parametr CURLOPT_TIMEOUT

Nastavuje maximální dobu čekání na stažení obsahu URL. Doba čekání je uvedena v sekundách (`int`).

8.3.4 Parametr CURLOPT_NOSIGNAL

Tento parametr nastavuje způsob vnitří komunikace o daných spojeních uvnitř knihovny. Je využit například při kontrole timeoutu spojení nastaveného pomocí `CURLOPT_TIMEOUT`.Tento parametr je důležité ve vícevláknových aplikacích nastavit na hodnotu 1 (`bool`).

8.3.5 Parametr CURLOPT_PORT

Nastavuje port (`int`), na který se chceme připojit.

8.3.6 Parametr CURLOPT_USERAGENT

Nastaví identifikaci (`*char`) programu pro volaný server. Je odeslána v hlavičce HTTP požadavku.

[5] CURLcode curl_easy_perform(CURL * handle);
[6] CURLcode curl_easy_setopt(CURL *handle, CURLoption option, parameter);

Kapitola 9

Balík programů autotools

Tento poměrně rozsáhlý balík programů a skriptů slouží především pro unifikovaný postup kompilace a nastavení její konfigurace. Z tohoto balíku bylo využito tří programů a to `autoscan`, `autoheader` a `autoconf`.

9.1 Program autoscan

Tento program dokáže z dostupných hlavičkových a zdrojových souborů v aktuálním adresáři zpracovat *definice*[1] a s těmi dále pracovat. Je to celkem schopný pomocník, který dokáže zjistit závislosti s mnoha knihovnami. Vytvoří soubor "`configure.scan`", ve kterém vytvoří základní strukturu maker. Tento soubor přejmenujeme na "`configure.in`" a následně editujeme několik základních hodnot, které dále zpracuje program `autoheader`.

9.2 Program autoheader

Jedná se o program, který vezme na vstupu soubor "`configure.in`", nebo "`configure.ac`" a jeho výstupem je klasický hlavičkový soubor[2]. Jeho název má přídavek ".in" za jménem souboru a obsahuje některé proměnné. Ty jsou při spuštění konfigurace zaměněny za skutečné hodnoty, zjištěné právě touto automatickou konfigurací a zároveň je soubor[3] pojmenován bez koncovky ".in"[4]. Program `autoheader` prochází "`configure.in`" řádek po řádku a zpracovává jednotlivá makra, která mohou být i klasickým *shell*[5] skriptem. V souboru musí být nastaveno několik základních maker, kde některá jsou po výstupu z `autoscan` označena textem `FIXME`. Jedná se především o `AC_PREREQ`, `AC_INIT`, `AC_CONFIG_SRCDIR`, `AC_CONFIG_HEADER`, `AC_CHECK_LIB`, `AC_CONFIG_FILES` a `AC_OUTPUT`. Existuje však mnoho dalších maker, která mohou být zavedena ihned po výstupu z programu `autoscan`. Jak vypadá soubor "`configure.in`" je uvedeno v příloze **A** na straně 28.

9.3 Vybraná makra programu autoheader

Zde uvedu několik základních maker, které byli využity v bakalářské práci.

[1] uvozené jako "`#define ...`"
[2] defaultně nastaveno na "`configure.h.in`"
[3] plus další uvedené v "`configure.in`"
[4] "`configure.h.in`" → "`configure.h`"
[5] program, který čte příkazy z terminálu/souboru a ty spouští

9.3.1 Makro AC_PREREQ

Označuje verzi `autoheader` potřebnou pro zpracování souboru.

9.3.2 Makro AC_INIT

Obsahuje jméno, verzi a kontaktní e-mail vztahující se k projektu.

9.3.3 Makro AC_COPYRIGHT

Obsahem je řetězec o právech na daný projekt. Je například součástí výpisu informací o projektu:

```
./configure -V
```

9.3.4 Makro AC_CONFIG_SRCDIR

Zde uvedeme jeden soubor, který je obsahem našeho projektu. Jeho účel je pro kontrolu, zda jsme ve správném adresáři.

9.3.5 Makro AC_CONFIG_HEADER

Zadáme hlavičkový soubor, do kterého chceme zpracovat výstup.

9.3.6 Makro AC_PREFIX_DEFAULT

Označuje adresář, ve kterém se hledají externí knihovny.

9.3.7 Makro AC_CHECK_LIB

Otestuje danou knihovnu, zda se v ní nachází požadovaná funkce.

9.3.8 Makra AC_CHECK_HEADER a AC_CHECK_HEADERS

Otestuje, zda jsou dostupné uvedené knihovny.

9.3.9 Makro AC_CHECK_FUNCS

Otestuje, zda jsou dostupné uvedené funkce.

9.3.10 Makro AC_SUBST

Zadáme, jaké další proměnné si přejeme nahrazovat v ".in" souborech. K jejich předchozímu nastavení může posloužit jednoduchý shell skript.

9.3.11 Makro AC_CONFIG_FILES

Uvedeme, ve kterých souborech se mají nahrazovat proměnné. Tyto soubory jsou při spuštění "`./configure`" a uvedení `AC_OUTPUT` přejmenovány tím způsobem, že se odstraní přípona ".in".

9.3.12 Makro AC_OUTPUT

Závěrečné makro. Provádí zmíněné nahrazení proměnných v `AC_CONFIG_FILES` souborech a jejich následné přejmenování.

9.3.13 Makra AC_MSG_ERROR a AC_MSG_NOTICE

Jedná se o makra pro výstup určité informace o stavu testování. Zatímco při použití makra `AC_MSG_ERROR` skript vypíše požadovanou informaci a skončí, při použití `AC_MSG_NOTICE` pouze vypíše informaci a pokračuje dále.

9.4 Program autoconf

Program autoconf vyrobí ze souboru "`configure.in`" spustitelný shellový skript pojmenovaný jako "`configure`". Ten se stará o správné nastavení proměnných, které zapíše do souborů uvedených v `AC_CONFIG_HEADER` a `AC_CONFIG_FILES`.

Kapitola 10

Kompilace, spuštění a testování

Zde bude v rychlosti shrnuto, jak s programem zacházet a dosažené výsledky.

10.1 Kompilace a instalace

Pro kompilaci je zapotřebí překladače `gcc` a nainstalovaná knihovna `libCURL`. Postupujeme takto:

- Rozbalíme soubor a přepneme se do vytvořeného adresáře.

  ```
  tar xzf url_caller.tar.gz
  cd url_caller
  ```

- Nakonfigurujeme instalaci.

  ```
  ./configure
  ```

- Zkompilujeme.

  ```
  make
  ```

- Nainstalujeme s oprávněním uživatele `root`.

  ```
  make install
  ```

Skript "`configure`" má několik volitelných parametrů:

- `LOCK_PATH` - určuje adresář umístění lockovacího `PID` souboru (`url_caller.lock`)
  default: `/var/lock/url_caller`
- `RUN_PATH` - určuje adresář kde budou rozmístěny pracovní adresáře (`pool`, `work`, `failed`, `lost`)
  default: `/var/url_caller`
- `PREFIX` - určuje prefix pro výslednou binárku (výsledek je v `$PREFIX/bin/url_caller`)
  default: `/usr/local`
- Příklad:

  ```
  ./configure LOCK_PATH=/var/lock
  ```

Pokud jsme konfiguraci neměnili, pak:

- Server spustíme pomocí:

  ```
  /usr/local/bin/url_caller
  ```

- Konfiguraci znovu načteme díky:

  ```
  kill -HUP ‘cat /var/lock/url_caller/url_caller.lock‘
  ```

- Server ukončíme příkazem:

  ```
  kill -KILL ‘cat /var/lock/url_caller/url_caller.lock‘
  ```

Pro zjednodušení situace byly v adresáři `skripts` vytvořeny skripty zajišťující tuto funkcionalitu.

10.2 Testování

Pro účely testování byl vytvořen jednoduchý klient, který pošle na zadaný server 1000 požadavků. Nalezneme ho po kompilaci pod názvem “`client`”. Příklad:

```
./client localhost 8011 ‘‘http://thor’’
```

Raná verze systému je v testovacím provozu přibližně od 12.12.2006 na serveru firmy *Mobilbonus s.r.o.* v datovém centru *Casablanca* v Praze. Do současné doby, z dostupných informací, server stále běží s průměrnou zátěží ∼10 požadavků za sekundu. Testovaná verze zvládla v plné zátěži ∼4000 požadavků za sekundu. Tato hodnota byla dosažena na notebooku při vytížení procesoru ∼25 %. Limitujícím faktorem je využití **ne** keep-alive soketů a tím pádem velká datová zátěž způsobená navazováním komunikace. Dosažené hodnoty však jsou více než dostačující a při použití gigabitové síťové karty se možnosti posouvají mnohem dále.

V současné době by měla být dostupná stabilní verze pro operační systém linux (`Gentoo`, `SuSe`, `Debian`). Pro stabilní běh je nutné zapnout `NPTL` vlákna. Cílovou distribucí bylo zvoleno `Gentoo`.

Kapitola 11

Závěr

Cílem projektu bylo vyvinout systém pro odesílání `URL` dotazů v programovacím jazyce `C`. Velkou část projektu tvořilo doprogramovávání jinak běžných funkcí, které při použití čisté `ANSI` normy nejsou dostupné.

Jak již bylo zmíněno, software byl primárně vyvíjen pro linuxovou distribuci `Gentoo`. Momentální cíl dalšího snažení je v stabilizování verze pod `FreeBSD`, případně pod dalšími UNIX-like systémy. Jako další rozšíření této práce by mohlo být obohacení o nové konfigurační volby a zapracování kompletní podpory certifikátů pro `HTTPS` spojení.

Systém již po relativně dlouhou dobu běží bez vážnějších problémů, s funkčností prozatimně dostačující jeho účelu.

Projekt se nachází na internetové adrese http://horimir.pinda.cz/ pod názvem *url_caller*. Zde bude možné v budoucnu stáhnout aktualizované verze. Program je šířen pod licencí `GNU/GPL`.

Literatura

[1] libCURL - the multiprotocol file transfer library. http://curl.haxx.se/libcurl/.

[2] The MIT license. http://www.opensource.org/licenses/mit-license.php.

[3] Zdrojové kódy k webovému serveru Apache. http://httpd.apache.org/.

[4] Zdrojové kódy k webovému serveru lighttpd. http://www.lighttpd.net/.

[5] Zdrojové kódy Linuxového jádra. http://www.kernel.org/.

[6] Marc J. Rochkind. *Advanced UNIX Programming, Second Edition.* Addison-Wesley Professional, 2004. ISBN 0131411543.

[7] W. Richard Stevens. *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications.* Addison-Wesley UK, 1999. ISBN 0-13-081081-9.

[8] W. Richard Stevens. *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API.* Addison-Wesley, 2003. ISBN 0-13-141155-1.

Kapitola 12

Příloha A

Soubor ‘‘configure.in’’

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
AC_INIT(url_caller, 0.9.0.0, horimir@atlas.cz)
AC_COPYRIGHT(Shared under GNU/GPL licence)
echo ‘‘+--------------------+’’
echo ‘‘|     url_caller     |’’
echo ‘‘|    Made by Magik   |’’
echo ‘‘+--------------------+’’
AC_CONFIG_SRCDIR([cfg.c])
AC_CONFIG_HEADER([autoconf.h])
AC_PREFIX_DEFAULT(’/usr/local’)

# Checks for programs.
AC_PROG_CC
AC_PROG_CXX

# Checks for libraries.
# FIXME: Replace ‘main’ with a function in ‘-lpthread’:
AC_CHECK_LIB([pthread], [pthread_detach], ,
[AC_MSG_ERROR([Sorry missed function, cant’t build])])

# Checks for header files.
AC_HEADER_DIRENT
AC_HEADER_STDC
AC_CHECK_HEADERS([arpa/inet.h fcntl.h limits.h netdb.h netinet/in.h stdlib.h
string.h strings.h sys/socket.h syslog.h unistd.h], ,
[AC_MSG_ERROR([Sorry missed header file, cant’t build])])

OTHER_INCLUDE_PREFIX=‘‘/usr/local/include’’
AC_CHECK_HEADER([curl/curl.h], , [AC_MSG_NOTICE([Checking for other
include prefix])
AC_CHECK_HEADER([$OTHER_INCLUDE_PREFIX/curl/curl.h], ,
[AC_MSG_ERROR([Sorry missed header file, cant’t build])])])

# Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
AC_TYPE_SIZE_T
```

```
40
41 # Checks for library functions.
42 AC_FUNC_CLOSEDIR_VOID
43 AC_FUNC_FORK
44 AC_FUNC_MALLOC
45 AC_FUNC_REALLOC
46 AC_TYPE_SIGNAL
47 AC_FUNC_STAT
48 AC_CHECK_FUNCS([atexit gethostbyaddr gethostbyname inet_ntoa memset
49 socket strerror], , [AC_MSG_ERROR([Sorry missed function, cant't build])])
50
51 test -z $LOCK_PATH && LOCK_PATH=``/var/lock/url_caller'';
52 test -z $RUN_PATH && RUN_PATH=``/var/url_caller'';
53 AC_SUBST([LOCK_PATH RUN_PATH])
54
55 AC_CONFIG_FILES([Makefile daemonize.h dirs.h skripts/run.sh skripts/stop.sh])
56 AC_OUTPUT
57 echo '+----------------------------+'
58 echo '|Ready to compile, type ``make''|'
59 echo '+----------------------------+'
```

Kapitola 13

Příloha B

Soubor ‘‘/etc/url_caller.conf’’

```
[URL ‘‘thor’’]
GROUP ‘‘thor’’

[URL ‘‘http://thor’’]
GROUP ‘‘thor’’

[GROUP ‘‘thor’’]
MAX_RECALL 2
TIMEOUT 2

[GROUP ‘‘default’’]
TIMEOUT 10
MAX_RECALL 3
#dsjhfieu

[CONFIG]
PORT 8011
USER_NAME ‘‘nobody’’
GROUP_NAME ‘‘nogroup’’

[DEFAULT]
GROUP ‘‘default’’
#do not uncoment last line
```